Fast and Accurate Computation of Sum and Dot Product

Takeshi Ogita (JST/Waseda Univ.)

joint work with
Siegfried M. Rump (TUHH) and Shin'ichi Oishi (Waseda Univ.)

Dagstuhl Seminar, Sep. 29, 2005

Paper available in SISC 26:6 (2005), 1955–1988.

Contents

Outline

The quality of a result obtained by floating point arithmetic on computers sometimes happens to be poor because of the rounding errors.

Purpose To compute summation $\sum_{i=1}^{n} x_i$ and dot product $x^T y = \sum_{i=1}^{n} x_i\, y_i$ accurately by floating point arithmetic.

$\Longrightarrow$ one of the most basic task in scientific computing

Background

There are a number of algorithms to compute a summation or a dot product. With respect to the *result* of algorithm we can grossly distinguish three different approaches:

i) heuristic/numerical evidence for improvement of result accuracy,

ii) result accuracy as if computed in higher precision,

iii) result with prescribed accuracy.

Our methods belong to the second class.

Disadvantages of previous methods

- sorting of input data is necessary, either

  i) by absolute value or,
  ii) by exponent,

- the inner loop contains branches,

- besides working precision, some extra (higher) precision is necessary,

- access to mantissa and/or exponent is necessary.

Our approach

- Only using ordinary (one working precision) floating point addition, subtraction and multiplication
- Based on Knuth's algorithm and Veltkamp-Dekker's algorithm
- No branch, no access to mantissa nor exponent

$\Longrightarrow$ Extra higher precision or special hardware is NOT necessary
$\Longrightarrow$ From these properties, our algorithms are very fast in terms of not only flops but also execution time

Error-free transformation

$\mathbb{F}$: a set of t-bit (working precision) floating point numbers
$\mathbf{u} := 2^{-t}$: relative precision of floating point arithmetic
For example, $t = 53$ in IEEE 754 double precision format ($\mathbf{u} = 2^{-53}$)

For $a, b \in \mathbb{F}$, we can obtain the following $x, y \in \mathbb{F}$:

Knuth (1969) $[x, y] = \mathbf{TwoSum}(a, b) \quad \Rightarrow \quad x + y = a + b$

Dekker (1971) $[x, y] = \mathbf{TwoProduct}(a, b) \quad \Rightarrow \quad x + y = a \times b$

Error-free! (no loss of information)

TwoSum (Knuth, 1969)

$$[x, y] = \textbf{TwoSum}(a, b) \quad \Rightarrow \quad a + b = x + y \quad (|y| \leq \mathbf{u}|x|)$$

$\oplus, \ominus$: floating point addition and subtraction

function $[x, y] = \textbf{TwoSum}(a, b)$
$x = a \oplus b;$
$c = x \ominus a;$
$y = (a \ominus (x \ominus c)) \oplus (b \ominus c);$

Computational cost: 6 flops

TwoSum' (Dekker, 1971)

$$[x, y] = \textbf{TwoSum}'(a, b) \quad \Rightarrow \quad a + b = x + y \quad (|y| \leq \mathbf{u}|x|)$$

function $[x, y] = \textbf{TwoSum}'(a, b)$
$x = a \oplus b;$
if $|a| \geq |b|$
  $y = b \ominus (x \ominus a);$
else
  $y = a \ominus (x \ominus b);$

3 flops + taking 2 absolute values + 1 comparison
$\Longrightarrow$ Computational speed slows down

Table 1: Comparison of measured computing time for Knuth's **TwoSum** with Dekker's **TwoSum**′

n	ratio (Dekker/Knuth)
100	1.342
1000	2.210
10000	3.046

CPU: Pentium 4 (3.46GHz), GNU Compiler Collection 3.4.2 (g77)

$\Longrightarrow$ We prefer to use Knuth's **TwoSum**.

Split (Dekker, 1971)

$$[a_H, a_L] = \textbf{Split}(a) \quad \Rightarrow \quad a = a_H + a_L \quad (|a_L| \le 2^{\lfloor \frac{t}{2} \rfloor} |a_H|)$$

$\otimes$: floating point multiplication

function $[a_H, a_L] = \textbf{Split}(a)$
$c = \texttt{factor} \otimes a;$ % $\texttt{factor} = 2^{\lceil \frac{t}{2} \rceil} + 1$
$a_H = c \ominus (c \ominus a);$
$a_L = a \ominus a_H;$

4 flops

TwoProduct (Veltkamp [Dekker, 1971])

$$[x, y] = \textbf{TwoProduct}(a, b) \quad \Rightarrow \quad a \times b = x + y \quad (|y| \leq \mathbf{u}|x|)$$

function $[x, y] = \textbf{TwoProduct}(a, b)$
$x = a \otimes b;$
$[a_H, a_L] = \textbf{Split}(a); \quad \%\ a_H + a_L \leftarrow a$
$[b_H, b_L] = \textbf{Split}(b); \quad \%\ b_H + b_L \leftarrow b$
$y = a_L \otimes b_L \ominus (((x \ominus a_H \otimes b_H) \ominus a_L \otimes b_H) \ominus a_H \otimes b_L);$

17 flops

TwoProductFMA

FMA: Fused Multiply-Add

By **FMA**(a, b, c), $d := a * b + c$ can be calculated in 1 flop. Result d is rounded to the nearest floating point number.

function $[x, y] =$ **TwoProductFMA**(a, b)

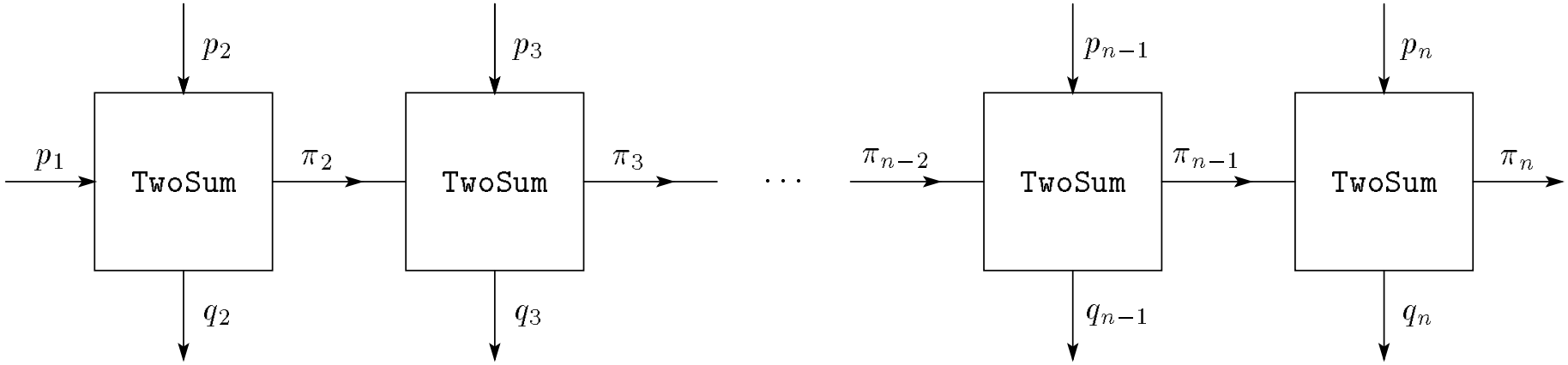
$x = a \otimes b;$

$y =$ **FMA**$(a, b, -x);$

17 flops $\Longrightarrow$ 2 flops

Error-free vector summation (1)

For $p \in \mathbb{F}^n$, we can cascade **TwoSum** as follows:



VecSum

$$p' = \textbf{VecSum}(p) \quad \Rightarrow \quad \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i$$

(so-called distillation algorithm)

function $p = \textbf{VecSum}(p)$
for $i = 2 : n$
$[p_i, p_{i-1}] = \textbf{TwoSum}(p_i, p_{i-1})$;

$6(n-1)$ flops

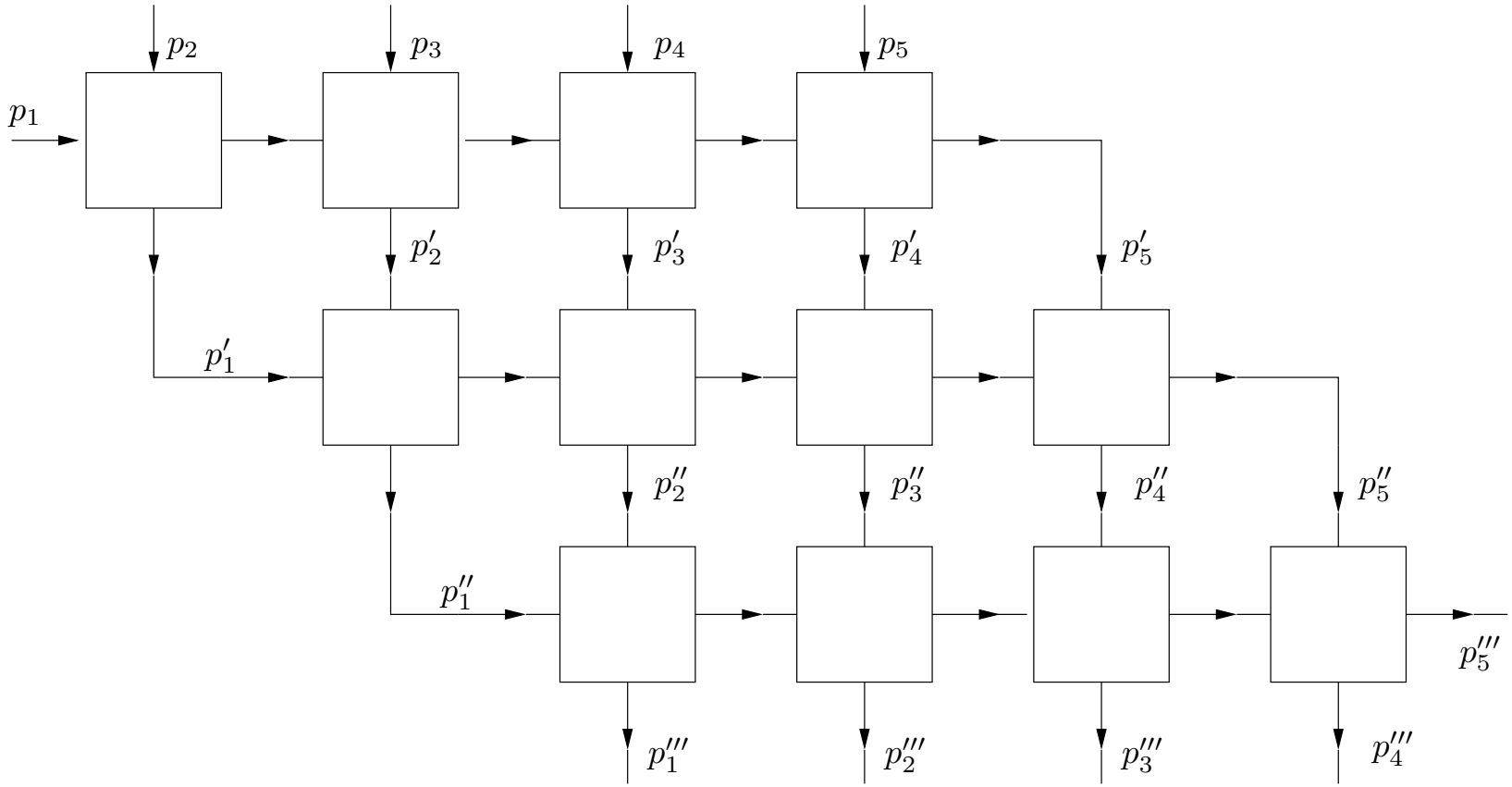
Error-free vector summation (2)

VecSum can iteratively be applied:

$$\sum_{i=1}^{n} p_i \xrightarrow{\textbf{VecSum}} \sum_{i=1}^{n} p'_i \xrightarrow{\textbf{VecSum}} \sum_{i=1}^{n} p''_i \xrightarrow{\textbf{VecSum}} \cdots$$

$$\Longrightarrow \quad \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i = \sum_{i=1}^{n} p''_i = \cdots$$

Error-free!



Example: $n = 5$, $K = 4$.

SumK

$$\textbf{res} = \textbf{SumK}(p, K) \quad \Rightarrow \quad \textbf{res} \sim \sum_{i=1}^{n} p_i$$

function $\textbf{res} = \textbf{SumK}(p, K)$
for $k = 1 : K - 1$
$p = \textbf{VecSum}(p);$
end
$\textbf{res} = p_n \oplus \mathrm{fl}\left(\sum_{i=1}^{n-1} p_i\right);$

$(6K - 5)n$ flops

Error analysis for SumK

$$\gamma_n := \frac{n \cdot \mathbf{u}}{1 - n \cdot \mathbf{u}} \quad (\mathbf{u} := 2^{-53} \text{ in IEEE 754 double precision})$$

$$\mathrm{cond}(\sum p_i) := \frac{\sum |p_i|}{|\sum p_i|} \quad \text{(condition number of summation)}$$

Denote by **res** $\in \mathbb{F}$ the result obtained by **SumK**, then

$$\frac{|\mathbf{res} - \sum p_i|}{|\sum p_i|} \leq \mathbf{u} + 3\gamma_{n-1}^2 + \gamma_{2n-2}^K \mathrm{cond}(\sum p_i).$$

$\Longrightarrow$ Basically, relative error bound is $\mathbf{u} + c_n \mathbf{u}^K \mathrm{cond}(\sum p_i)$

$\Longrightarrow$ K-fold working precision arithmetic

Accurate dot product

We can extend the discussion to dot product using **TwoProduct**.

For x, $y \in \mathbb{F}^n$, using $[h_i, r_i] = \mathbf{TwoProduct}(x_i, y_i)$

$$\sum_{i=1}^{n} x_i\, y_i = \sum_{i=1}^{n} (h_i + r_i) \quad \left(= \sum_{i=1}^{2n} t_i \right)$$

Error-free! (provided no underflow occurs)

Dot2

function res $=$ **Dot2**(x, y)
$[p, s] =$ **TwoProduct**(x_1, y_1);
for $i = 2 : n$
  $[h, r] =$ **TwoProduct**(x_i, y_i);
  $[p, q] =$ **TwoSum**(p, h);
  $s = s \oplus (q \oplus r)$;
end
res $= p \oplus s$;

$25n$ flops

Error analysis for Dot2

$x, y \in \mathbb{R}^n \quad \Rightarrow \quad$ Condition number of $x^T y$ is defined by

$$\mathrm{cond}(x^T y) := \frac{2|x^T|\ |y|}{|x^T y|}.$$

Denote by **res** $\in \mathbb{F}$ the result obtained by **Dot2**, then

$$\left| \frac{\mathbf{res} - x^T y}{x^T y} \right| \leq \mathbf{u} + \frac{1}{2}\gamma_n^2 \mathrm{cond}\,(x^T y).$$

$\Longrightarrow \mathbf{u} + c_n \mathbf{u}^2 \mathrm{cond}(x^T y)$ (doubled working precision arithmetic)

DotK

function res $=$ **DotK**(x, y, K)
$[p, r_1] =$ **TwoProduct**(x_1, y_1);
for $i = 2 : n$
  $[h, r_i] =$ **TwoProduct**(x_i, y_i);
  $[p, r_{n+i-1}] =$ **TwoSum**(p, h);
end
$r_{2n} = p$;
res $=$ **SumK**$(r, K-1)$;

$(12K + 1)n$ flops

Error analysis for DotK

Denote by **res** $\in \mathbb{F}$ the result obtained by **DotK**, then

$$\frac{|\mathbf{res} - x^T y|}{|x^T y|} \leq \mathbf{u} + 2\gamma_{4n-2}^2 + \frac{1}{2}\gamma_{4n-2}^K \mathrm{cond}(x^T y).$$

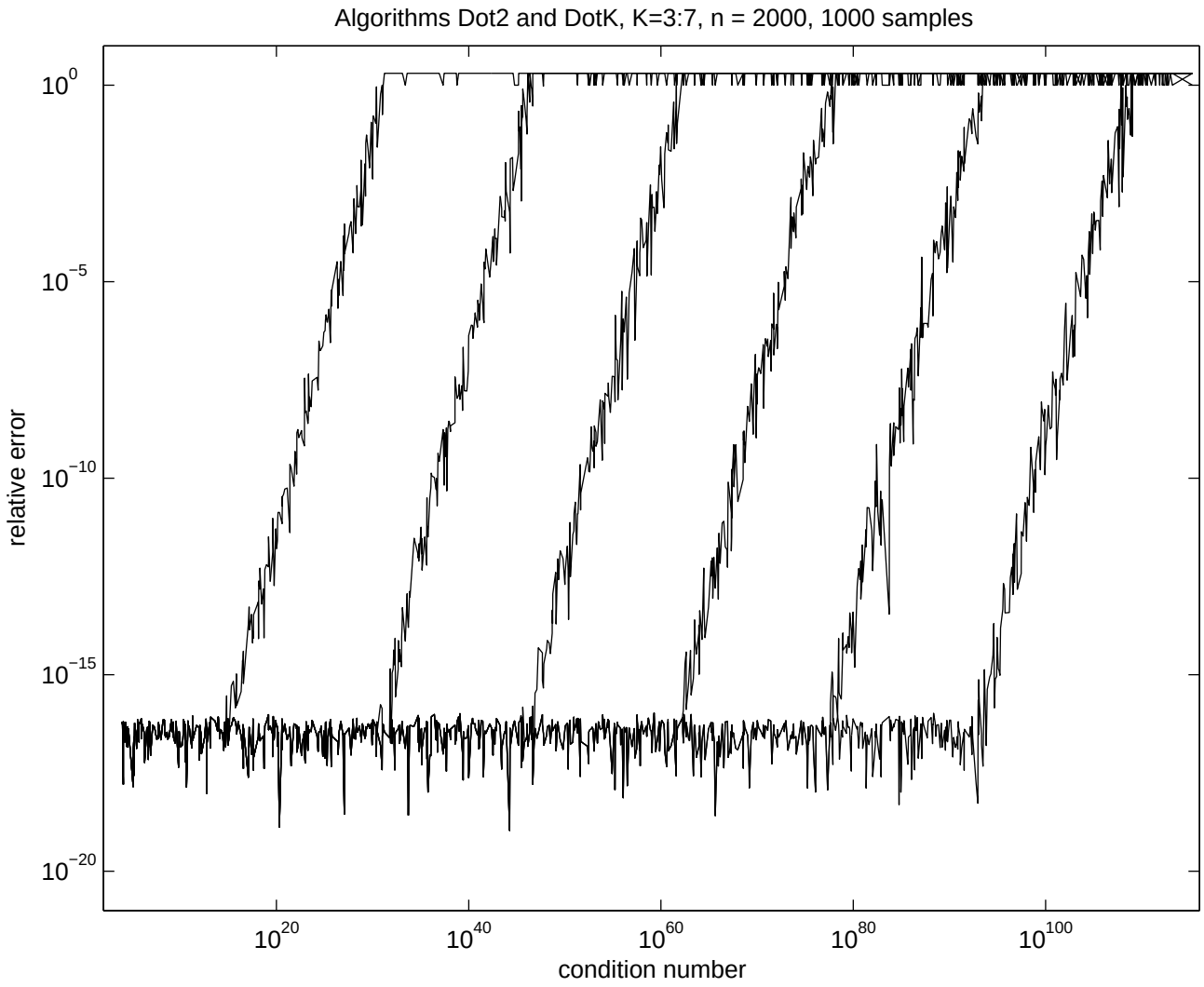
$\Longrightarrow$ Basically, relative error bound is $\mathbf{u} + c_n \mathbf{u}^K \mathrm{cond}(x^T y)$

$\Longrightarrow$ K-fold working precision arithmetic

Numerical example (1)

Let us evaluate accuracy of **Dot2** and **DotK**.

- $n = 2000$ (1000 samples)
- Elements of $x, y \in \mathbb{F}^n$: random numbers in $[-1, 1]$ s.t. $\mathrm{cond}(x^T y)$ is from 1 to 10^{120}
- Plot the relative error $\dfrac{|\mathbf{res} - x^T y|}{|x^T y|}$ where **res** is the result of **Dot2** or **DotK** $(K = 3 : 7)$


Algorithms Dot2 and DotK, K=3:7, n = 2000, 1000 samples
relative error
condition number
10^{0}
10^{-5}
10^{-10}
10^{-15}
10^{-20}
10^{20}
10^{40}
10^{60}
10^{80}
10^{100}

Numerical example (2)

Let us evaluate speed of matrix-vector product using **Dot2**.

- n: $100, 200, \ldots, 3000$
- For $A \in \mathbb{F}^{n \times n}, x, b \in \mathbb{F}^n$, $y = Ax - b$ is calculated
- Display the ratio of the measured computing time of our routine compared to that of the BLAS routine **DGEMV** over all dimensions

Table 2: Measured ratio of computing time for matrix-vector residual

env.	BLAS	**Dot2**	XBLAS (F)	XBLAS (C)
$*$)	1	4.8	9.3	11.3
theor.	1	12.5	18.5	18.5

$*$) Pentium IV 2.53 GHz, Fortran (CVF 6.6C)
C (MS VC++ 6.0), BLAS: Intel MKL 7.0

Application to linear systems

Let us consider $Ax = b$ with $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$. Denote by x^0 an approximate solution obtained by LU decomposition ($\frac{2}{3}n^3$ flops).

Then, **Dot2** is suitable for iterative refinement:

1. Calculate residual $\tilde{r} \sim b - Ax^\ell$ using **Dot2** ($\mathcal{O}(n^2)$ flops)
2. Solve $Ay = \tilde{r}$ using LU factors ($\mathcal{O}(n^2)$ flops)
3. Update $x^{\ell+1} = x^\ell + \tilde{y}$

In addition, this can be combined with self-validating method for linear systems. We use INLTAB, a fast interval toolbox for MATLAB.

Table 3: Results with/without residual iteration ($n = 1000$)

$\mathrm{cond}(A)$	10^5	10^9	10^{13}	LU
max. rel. err. x^0	4.7e-12	2.8e-08	1.9e-04	
max. rel. err. x^ℓ	1.8e-16	1.8e-16	1.8e-16	
number of iterations ℓ	3	3	5	
ratio computing time	1.38	1.38	1.65	1

Pentium IV 2.53 GHz, Matlab 6.5, INTLAB 4

Remark: For larger dimensions, the additional cost (including Matlab's interpretation overhead) relatively decreases.

Conclusion

- Develop and analyze accurate algorithms for sum and dot product
- Fast in terms of not only flops but also measured computing time
- Only using one working precision (portable and scalable)